Refactoring the Web Interface

John R. Douceur[†], Jon Howell[†], Bryan Parno[†], Michael Walfish[‡]
[†]Microsoft Research [‡]UT-Austin

1 The IRON properties of the web

The web browser, which originated as a simple viewer for displaying static web pages, has evolved into an operating system for executing web applications. The quantity, diversity, and capability of web applications have grown dramatically, such that many modern web applications have begun to rival the breadth and functionality of desktop applications. What has fueled this trend? Why do users find a browser to be a better application platform than a traditional operating system?

Somewhere within the amalgamation of standards and conventions that define what it means to be a web application, there must be some key properties that make such applications particularly attractive to users. We argue that these important properties are unrelated to most of the de facto web API, including HTML, DOM, CSS, GIF, JPEG, PNG, JavaScript, etc. In other words, an entirely different set of web standards could be just as attractive and successful, as long as it were to maintain a particular set of core properties.

In particular, we posit that web apps are attractive because they are *isolated*, *rich*, *on-demand*, and *networked*:

- Isolated: Web applications cannot unilaterally affect other applications, so they are safe to try.
- Rich: Web applications are visually appealing, interactively responsive, and semantically powerful.
- On-demand: Web applications do not require installation or OS configuration, so they are easy to test drive and easy to point others to.
- Networked: Web applications make use of resources on the web, so they can access and integrate a growing and up-to-date set of disparate content.

We argue that these properties—which we call the IRON properties—are individually *necessary* to preserve the attractiveness of the current web. If web apps were not isolated or on-demand, the increased risk or burden of trying out a new app would reduce its rate of proliferation. If they were not networked, many of today's most interesting web apps (online maps, electronic commerce, cloud storage, etc.) could not function. And although the early static web was not very rich, the introduction of client-side execution (via JavaScript) was needed to enable virtually every web app in use today.

We further believe that these properties are jointly *sufficient* to provide the user experience that makes web applications attractive to users. To demonstrate this, the Zoog project at Microsoft Research is constructing a minimal execution platform that satisfies the IRON properties. Our intent is to show that this platform can support the entire set of applications that exist on today's web.

2 Weaknesses of the current web API

Although the IRON properties are what makes web applications attractive, the current web API actually weakens all four of these properties.

Web apps are *not strongly isolated*, because the web API is very broad. Not only does the API include rendering interfaces for the suite of HTML standards (DOM, CSS) and an execution interface for JavaScript, but it also de facto includes interfaces to common image formats (GIF, JPEG, PNG) and popular plug-ins (Flash, JVM). This broad API is implemented via a large trusted computing base (TCB), which has evinced numerous security vulnerabilities and exploits [7], weakening the degree to which web applications are isolated.

Web applications are undoubtedly rich, but they are far *less rich* than desktop applications. In large part, this is because desktop apps can build on a huge volume of existing code and libraries, written in arbitrary languages, and built using a wide variety of toolchains. By contrast, only a small fraction of legacy code is written in or translatable to a web-standard language such as JavaScript, Flash bytecode, or JVM bytecode [5].

The web API enables applications to be richer by taking advantage of plug-ins, such as ActiveX controls and runtimes for Flash and Java. However, this makes web apps *less on-demand*, because these plug-ins require regular updating and patching, which requires users to make configuration decisions since the plug-ins are part of the TCB. Attempts to use plug-ins to deploy entirely new runtimes, such as Silverlight, suffer from low uptake because they require user action beyond merely clicking on a link. The on-demand nature of web apps is also weakened by browser incompatibilities that inevitably arise among different implementations of a complex API [6].

Web applications are *restrictively networked* by the web API's policy rules. The same-origin policy (SOP) prevents a web app from exchanging content with any site other than the app's origin site. (Displaying images and executing scripts are permitted exceptions, although reading the image or script content is not.) These rules arose to prevent attacks that leverage a browser's privileged location to compromise another machine, but their realization inhibits interesting classes of networked applications, most notably peer-to-peer applications.

These weaknesses have been noted before, and various research efforts have attempted to address them. OP [4], IBOS [9], Chrome [8], and Gazelle [10] improve the ability of a browser to keep applications isolated, by partitioning browser functionality to reduce the TCB size and to limit unintended cross-application dependencies.

Xax [5] and Native Client [11] are plug-in models that enable applications to become richer without becoming less on-demand, by executing native-code plug-ins inside isolated sandboxes. Consent protocols [1, 2] reduce the limitations of the SOP by providing safe means for web applications to communicate with non-origin servers.

Although these efforts have made significant progress, their potential is limited by the constraints of the current web API. Understandably, there is reluctance to change such a popular API, because of legacy concerns and because of developer familiarity with the existing API. However, we argue that the biggest problem is not so much changing the web API but rather disentangling two concepts that are tightly coupled (and, moreover, conflated) in the term "Application Programming Interface".

3 Deconstructing the web API

When a developer programs a desktop application, she writes code to implement application-specific behavior. This code employs both OS system services and also library components linked into the application. From the developer's point of view, she is writing the app to talk to a *Developer Programming Interface* (DPI). When a client OS executes an application, it executes instructions that occasionally call system services. The client OS is oblivious to calls from the application to its linked-in libraries. From the client's point of view, it is executing an app that is talking to a *Client Execution Interface* (CEI). For desktop applications, the difference between DPI and CEI can vary significantly depending on how much library code the developer chooses to include in the app.

For web applications, the DPI and CEI are commonly identical[1]: The developer writes her app to manipulate HTML via the DOM and CSS, using code written in JavaScript or Flash/Java/Silverlight, presenting images in GIF, JPEG, or PNG format. The code that implements this functionality is resident on the client, so the same interface is used by the client to execute the application.

There is no need for these two interfaces to be identical; the CEI merely needs sufficient functionality to satisfy the IRON properties and to support code that implements the DPI. Otherwise, the CEI should be as small as possible: A small CEI makes applications more strongly isolated because it reduces the size of the shared TCB, thereby minimizing opportunities for security vulnerabilities across applications. A CEI that enables native-code execution, a la Xax or Native Client, supports richer applications, because it allows the reuse of extant code, libraries, and toolchains. With a small CEI, all rich code is outside the TCB, so even applications with new rich functionality can be loaded without involving user configuration decisions, thus keeping them on-demand.

[1] Although toolkits such as GWT provide a higher-level DPI.

To support networked applications, we argue that each application should have a raw IP pipe to the internet, and this connection should be logically outside any firewall. It is clear that this removes the awkward SOP restrictions of the current web API, but one might reasonably be concerned that this proposal may increase a malicious application's ability to harm (1) other apps on the same machine, (2) other machines on the same subnet, or (3) other machines on the internet. We address each in turn.

Other applications on the same machine are strongly isolated from the malicious app, because our small CEI has a very narrow interface that we posit is easy to secure. The only means for two apps to communicate is via their network connections, so the malicious app's only avenue for attack is sending packets to the target app. Thus, the case of another app on the same machine reduces to the case of another machine on the internet.

Other machines on the same subnet might seem particularly vulnerable to non-SOP-restricted communication, because the SOP was developed to prevent attacks that leverage a browser's privileged location. However, in our proposal, each web application's IP pipe is logically outside any firewall, so the browser is not in a privileged location. For legitimate web apps that need access to enterprise resources behind a firewall, each app must establish a separate authenticated connection to the enterprise server, analogous to the use of VPNs to provide access to enterprise resources for remote clients.

Lastly, other machines on the internet are no more vulnerable to a malicious app when it runs on a client than when it runs on the attacker's home machine. An attacker *does* obtain additional resources from the client, with which it can launch DDoS attacks. However, the client can enforce correct source addresses in outgoing packets, and it can include source-side mechanisms [3] to protect against DDoS attacks.

References

[1] Adobe. Cross-domain policy file specification. `http://www.adobe.com/devnet/articles/crossdomain_policy_file_spec.html`.

[2] I. Fette. The WebSocket protocol. `http://tools.ietf.org/html/draft-ietf-hybi-thewebsocketprotocol`, 2011.

[3] J. M. Gregory, G. Prier, and P. Reiher. Attacking DDoS at the source. In *IEEE ICNP*, 2002.

[4] C. Grier, S. Tang, and S. T. King. Secure web browsing with the OP web browser. In *IEEE Symp. on Security and Privacy*, 2008.

[5] J. Howell, J. R. Douceur, J. Elson, and J. R. Lorch. Leveraging legacy code to deploy desktop applications on the web. In *OSDI*, 2008.

[6] E. Kıcıman and B. Livshits. AjaxScope: A platform for remotely monitoring the client-side behavior of Web 2.0 applications. In *SOSP*, 2007.

[7] NIST Vulnerability Database. http://nvd.nist.gov/nvd.cfm.

[8] C. Reis and S. D. Gribble. Isolating Web Programs in Modern Browser Architectures. In *ACM EuroSys*, 2009.

[9] S. Tang, H. Mai, and S. T. King. Trust and Protection in the Illinois Browser Operating System. In *OSDI*, 2010.

[10] H. J. Wang, C. Grier, A. Moshchuk, S. T. King, P. Choudhury, and H. Venter. The multi-principal OS construction of the Gazelle web browser. In *USENIX Security Symposium*, 2009.

[11] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar. Native client: A sandbox for portable, untrusted x86 native code. In *IEEE Symposium on Security & Privacy*, 2009.